Chapter 13

STACK-BASED BUFFER OVERFLOWS IN HARVARD CLASS EMBEDDED SYSTEMS

Kristopher Watts and Paul Oman

Abstract Many embedded devices used to control critical infrastructure assets are based on the Harvard architecture. This architecture separates data and program memory into independent address spaces, unlike the von Neumann architecture, which uses a single address space for data and program code. Buffer overflow attacks in desktop and server platforms based on the von Neumann model have been studied extensively. However, buffer overflows in Harvard architectures have only just begun to receive attention. This paper demonstrates that stack-based buffer overflow vulnerabilities exist in embedded devices based on the Harvard architecture and that the vulnerabilities are easily exploited. The paper shows how the reversal in the direction of stack growth simplifies attacks by providing easier access to critical execution controls. Also, the paper examines defense techniques used in server and desktop systems and discusses their applicability to Harvard class machines.

Keywords: Embedded systems, Harvard architecture, buffer overflows

1. Introduction

The buffer overflow is a well-researched attack vector. However, most research has focused on high performance processors based on the von Neumann memory model. The von Neumann model uses a single address space for data and program code [4]. On the other hand, the Harvard architecture – which is widely used in embedded devices – separates data and program memory into independent address spaces [15]. The independent data address space allows the data segment to grow and be manipulated without regard to the location of the program memory. Because the address space in a von Neumann machine is shared by data and program code, the program must take steps to prevent data from interfering with program code. This paper examines the Harvard and von Neumann architectures and demonstrates how stack-based vulnerabilities in Harvard class machines render them vulnerable to buffer overflow exploits.

C. Palmer and S. Shenoi (Eds.): Critical Infrastructure Protection III, IFIP AICT 311, pp. 185–197, 2009.

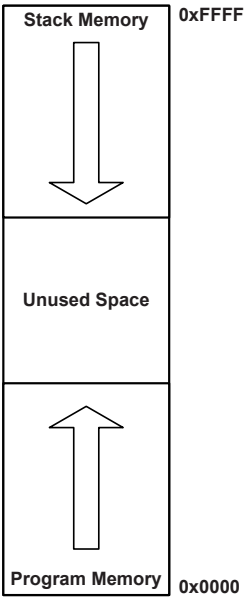

Figure 1. von Neumann memory model.

2. Stack-Based Buffer Overflows

Buffer overflow vulnerabilities have been documented since the early 1970's [2], but the most celebrated exploit involved the Morris worm of 1988 [11], which opened a Pandora's box of buffer overflows and other exploits. Stack-based buffer overflows were popularized by Levy (aka Aleph One) in his 1996 paper, "Smashing the Stack for Fun and Profit" [1]. A buffer overflow is an artifact of dynamic memory and occurs when a program attempts to write "too much" data to a specified location. The effect of writing excess data is that the extra data spills over the boundary of allocated space, possibly overwriting other important data. The effect of an overflow on a running program depends on where the allocated space exists within memory and how the space is used by the system and/or program. Levy's paper, which focused exclusively on von Neumann class architectures, examined overflows that occur in dynamic memory structures used by C-like languages.

A stack is a data structure with the property that the last item placed on the stack is the first item to be removed. A stack has two principal operations, Push and Pop. Push places a new item on top of the stack; Pop removes an item from the top of the stack.

The arrangement of a stack in memory varies according to the processor architecture and compiler implementation. A von Neumann machine almost always has a stack that starts in high address memory and grows into low address memory (Figure 1). On the other hand, a Harvard class machine

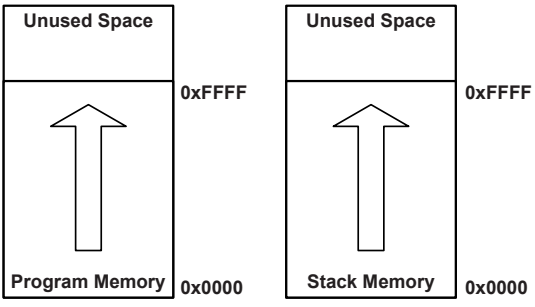

Figure 2. Harvard memory model.

usually has a stack that starts in low address memory and grows into high address memory [4, 15] (Figure 2).

Many modern programming languages use a stack to support dynamic calling mechanisms such as function calls. The stack stores the state of the machine prior to a dynamic jump. Specifically, whenever a function is invoked, the state of the machine is saved by pushing the processor registers on the stack along with bookkeeping items associated with the machine state (e.g., the current stack top pointer and return address for code execution). The result is a clean processor that is ready to execute the called function.

Each logical memory segment on a stack that is associated with an instance of a function is called a "frame." After a function has finished executing, the variables belonging to the previous function are popped off the stack and placed back in the processor; this effectively moves the function frame from the stack to the processor registers and status words. The return pointer is removed from the stack when a function exits; this pointer gives the address where execution resumes after the function has terminated. Thus, the execution sequence relies on the integrity of the stack. Corrupting the stack can cause a processor to execute unintended code, process invalid data or crash. Interested readers are referred to [8] for a discussion of stack-based buffer overflow exploits in von Neumann architectures.

The memory organization for an executing program is also dependent on the operating system and compiler. For instance, in an IA32 Linux system (von Neumann architecture), the executable code of a process usually starts at memory location `0x08000000` and the stack begins at `0xBFFFFFFF` and grows downward in memory [10]. Note, however, that the actual starting locations vary for different IA32 Linux versions.

Many environments, such as multitasking operating systems, use virtual memory systems to simplify the execution environment. The resulting "virtual address space" creates the illusion that each process can access the entire address space without interfering with other processes. These systems may complicate the physical memory organization, but simplify the operation of the stack from the point of view of an executing process. Virtual memory is

not discussed in this paper because it is typically not supported by Harvard architectures.

3. Harvard Architecture

The Harvard architecture is prevalent in small devices and embedded systems, but is relatively rare in higher capacity systems due to the cost of incorporating large amounts of integrated CPU memory. Harvard class microprocessors are used for low-power, high-reliability embedded applications such as microcontrollers, sensors and actuators. Examples include vehicle engine controllers, flight systems, mobile communication systems and remote sensing equipment. Embedded control devices are ubiquitous in critical infrastructure components and are becoming increasing common in consumer products. Security research, however, has historically focused on desktop and server environments, and has only recently turned its attention to embedded systems.

This paper focuses on the Intel 8051 Harvard class microprocessor [14]. Introduced by Intel in 1980 as a logical extension to its 8048 microprocessor, the Intel 8051 exemplifies all the characteristics of a Harvard class processor – a comparatively small instruction word, small program space and a minuscule data space. The original chip contains 4 KB of ROM and 128 bytes of RAM, both integrated into the chip.

We use the C8051F530 embedded system development kit from Silicon Laboratories, which is based on the Intel 8051 architecture. The development kit contains a complete board with an interface to the processor, several output ports and a Keil C compiler for the Intel 8051. The board also contains a JTAG port that provides direct access to ROM and RAM during processor execution (primarily for debugging purposes). The JTAG-based debugging system enables programmers to inspect the machine state during execution and to view the entire RAM contents. The board also contains an integrated universal asynchronous receiver transmitter (UART) that communicates with the processor and delivers data to the running program.

4. Stack Frames

As described earlier, the stack frame associated with a function call contains critical data that describes the status of the function and the flow of execution. The data placed in a frame is dependent on the programming language, operating system and compiler that are involved. The return pointer is a critical piece of data that is placed on the stack prior to each function call. The stack frame is an ideal location for the return pointer because each frame is associated with a single instance of a function. Compilers may store variables, processor registers and function parameters differently within a frame, but all compilers save the return pointer on the stack in some form or another.

The organization of stack frames is relatively consistent in compilers for von Neumann machines; however, compilers designed for Harvard architectures (e.g., Keil 8051 C compiler) organize stack frames in a non-conventional manner

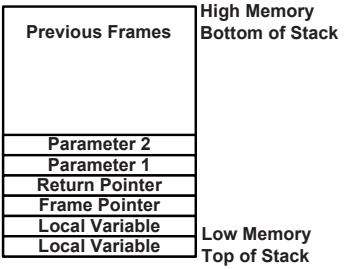

Figure 3. von Neumann stack organization.

[9]. Specifically, the Keil C compiler attempts to pre-allocate the stack space required for function calls. The stack frames built by Keil C are minimal and do not reside on top of the stack (unlike frames built by a GNU C compiler, for example [13]). Instead, Keil C frames exist lower in the stack and below the global variables. A frame pointer associated with a function call does not reside within a frame, but is instead pushed to the top of the stack.

The basic frame structure follows the order of operations outlined by a processor function call. Most modern processors automatically push the return pointer on the stack when a CALL instruction is encountered. The compiler must then produce a function header that saves the other processor registers. This task is left to the compiler instead of being integrated within the CALL instruction so that intelligent optimization routines can reduce the number of pushes and pops if registers are not needed during function execution.

Figure 3 presents an example of a frame produced by the GNU C compiler executing on an IA32 processor. Note that the first items pushed on the stack before a function call are the function parameters (placed on the stack by the calling function). The next items are the return pointer and the frame pointer of the previous function (note, however, that not all compilers and processors store the frame pointer). The last items placed on the stack are the local variables of the function.

It is important to note that, because the stack for the IA32 architecture starts in high memory and grows down, contiguous writes to local variables (e.g., a character array) begin writing in low memory and progress to high memory towards the locations of the return pointer and frame pointer. For example, the `strcpy()` function (which copies one array to another) creates a buffer overflow vulnerability in the IA32 architecture by overwriting the return pointer and causing the processor to return to a location different from what was intended. Exploit writers use such a misdirected return to gain control of an executing machine, essentially by modifying the execution path of a program.

Not all compilers and processors store the frame pointer. Harvard architectures produce a frame that is organized differently from that produced by a von Neumann machine. Early Harvard architectures rarely stored the frame pointer because the data space was small enough to preclude the need for additional reference points. Moreover, since the stack is in a separate address space, it

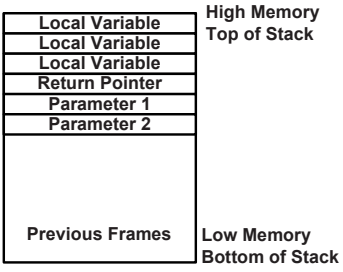

Figure 4. Harvard stack organization.

can begin at low memory and grow into high memory – the direction of stack growth is opposite to that for a von Neumann machine.

Figure 4 shows a typical frame in a Harvard architecture. The structure is similar to that of a von Neumann machine frame (with local variables on top of the stack). However, because a Harvard architecture stack grows from low memory to high memory, contiguous writes are made in the same direction as stack growth and, thus, do not approach the return pointer in the frame. As a result, it is impossible to overwrite a return pointer through a typical buffer overflow, except when the return pointers are placed above the local variables.

5. Buffer Overflow Exploits

Exploiting a buffer overflow vulnerability requires an intimate knowledge of stack behavior, execution procedures and the programming environment. The stack frame organization is especially significant because variables that are critical to the continued operation of a function may have to be overwritten to get to the targeted address. The change in the direction of stack growth in a Harvard architecture creates a more complex exploit environment because the buffer overflow moves away from the return pointer instead of towards it.

This section describes the process of overwriting a return pointer via a buffer overflow and examines how the process differs for the Harvard and von Neumann architectures. Our experiments used the C8051F530 development kit for the Intel 8051 microprocessor along with the Keil C compiler. The Keil C compiler employs several defensive techniques for inhibiting buffer overflow, which will be discussed in more detail later in this paper.

A simple terminal application connected to the UART (called the "Echo" program) was used to explore buffer overflows. The Echo program incorporates a UART and hardware timer drivers along with interface functions for gathering and sending data to the UART. The UART driver is an interrupt-driven system that transfers data to and from the UART port asynchronously. A simple echo routine is used to pull data from the UART buffers as a string terminated by a newline character. The function that pulls strings from the UART is similar to the `getline()` function in the GNU C library. After a complete string has been input, the string is copied into another buffer using the `strcpy()` function

```
Function func{
    char buff1[32];
    char buff2[16];
    while(1){
        ...
        GetStringFromUART(buff1);
        strcpy(buff2, buff1);
        PutStringToUART(buff2);
        ...
    }
}
```

Figure 5. Echo program pseudocode.

and the new buffer is sent back to the terminal. If a user were to connect to the embedded device through a terminal program (e.g., HyperTerminal), the program would simply display what the user types.

The GNU C library `strcpy()` function was implemented as a known vector for buffer overrun exploits because it insecurely copies one array to another – it does not perform any bounds checking. Interested readers are referred to [8] for a complete list of C library vulnerabilities and to [7] for a comparison of vulnerability detection tools.

Our Echo program uses a `func()` function to create two buffers of different sizes and then invokes `strcpy()` to insecurely copy data from a larger buffer to a smaller buffer – a classic coding error. The Echo program also contains an orphaned function, `owned()`, which continuously echoes the string "owned!" to the UART. The `owned()` function is considered to be orphaned because it is not called by any other function and is unreachable via normal execution paths. Our test program uses a stack-based buffer overflow to redirect the execution of the Echo program from `func()` to `owned()`. Once the redirection occurs, execution never exits `owned()`, and an endless stream of "owned!" text is produced, which indicates that the exploit is successful.

Figure 5 shows the Echo program. Variables `buff1` and `buff2` are defined as global space for arrays of size 32 and size 16, respectively. Thus, a buffer of size 32 is copied to a buffer of size 16 without bounds checking. The result is that up to 15 bytes of data can be written beyond the bounds of `buff2` on the stack plus a null terminator. The overwritten data can contain important control identifiers that dictate the flow of execution and data passed to other functions. As shown in Figure 3, a successful buffer overflow in the frame of a von Neumann machine enables an attacker to access all the variables defined after the buffer along with the return pointers and frames located deeper in the stack. The exact opposite is true for Harvard architectures. The reversed direction of code growth means that an attacker has access only to the data defined after the overflowed buffer. Note, however, that subtle nuances in stack implementation may create exploitable vulnerabilities.

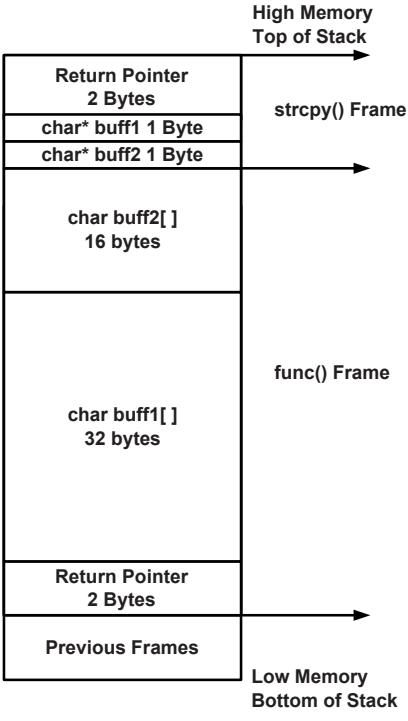

Figure 6. Mapping a buffer overrun exploit.

In general, it is relatively difficult to successfully gain control of a process without causing an illegal instruction or reference to unallocated memory. Figure 6 shows a stack representation (similar to the GNU C compiler) for the Atmega AVR, which is also a Harvard architecture. The global variables are placed at the base of the stack before any functions are allocated.

Note that `strcpy()` is called by `func()` and is passed two parameters with the addresses of the source (`buff1`) and destination (`buff2`) buffers. The `strcpy()` function repeatedly references the parameters within its frame during the copying process; thus, any corruption of the values causes the function to write data to incorrect locations. In order to gain control of the process, an attacker must write a value into the two bytes containing the return pointer for `strcpy()`. This causes `strcpy()` to return to a location of the attacker's choosing (in our case, the orphaned function that repeatedly outputs "owned!"). The challenge is to get to the location of the return pointer without corrupting the parameters passed to `strcpy()` by including the proper values for the two parameters in the source buffer. The overflow writes valid data to the two parameters used by `strcpy()`; the overwritten parameters must contain the correct values that allow `strcpy()` to proceed normally and successfully write to the memory locations containing the return pointer.

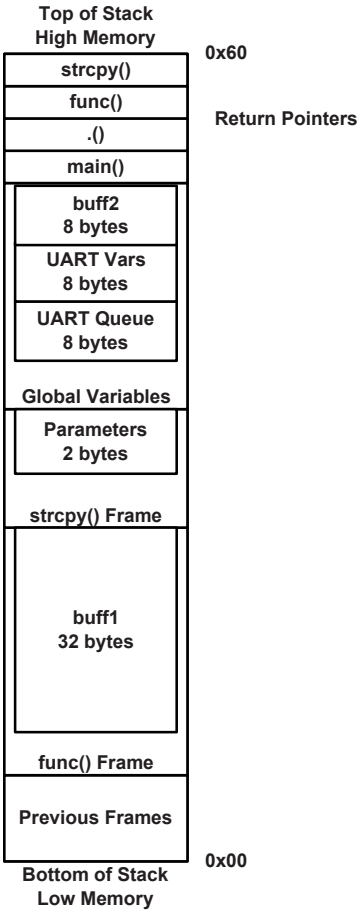

Figure 7. Stack structure of Echo program.

Figure 6 suggests that a Harvard architecture that places the return address on top of the stack should be easy to exploit. However, this is difficult in practice because of the efforts taken by compiler writers to organize memory effectively. The Keil C compiler for the Intel 8051 is an excellent example of a non-standard stack arrangement devised to simplify the use of limited RAM [9]. The Keil C compiler does not include return pointers in stack frames, but instead rearranges global variables in a way that essentially pre-allocates space for function variables. It moves the global variables towards the top of the stack, above the space where local function variables reside and, instead of placing return pointers inside stack frames, aggregates them at the very top of the stack above the global variables. Thus, stack frames constructed by the Keil C compiler are not reentrant.

Figure 7 shows the actual stack mapping for the Echo program. Because of the arrangement of global variables by the Keil C compiler, an overflow of `buff2`

allows direct access to the return pointers. An attacker can directly access the uppermost return pointer without regard for the other return pointers that may be destroyed – this is because the uppermost return pointer is the first to be loaded. In fact, the attacker can gain immediate control merely by loading the uppermost return pointer with a valid location.

6. Exploit Payloads

As described above, it is possible to conduct a successful exploit on a Harvard architecture embedded system. However, the "payload" of the exploit is an issue that deserves consideration. Specifically, what would an attacker hope to execute that is not already in the ROM of the embedded system? Developing a "worthwhile" exploit is more difficult than in a traditional von Neumann architecture where the processor makes no distinction between data and code.

An exploit on a von Neumann system can deliver custom code within the overflow string and jump to that code, enabling the attacker to insert functionality that was never present in the original program. At first glance, this does not seem possible in a Harvard architecture because the code is "frozen" in ROM or flash memory, and the processor only manipulates data in the data address space. The memory separation effectively eliminates the ability of an attacker to inject custom code into the execution stream. The attacker must instead use the functionality already present in the system. However, there are some well-researched analogs to building an effective payload for Harvard architectures. For example, it is possible to create a buffer overflow that performs a `return-to-libc` attack, which then injects entirely new code into the data segment of an Atmega based wireless sensor [5].

Non-executable stacks have been implemented in BSD, Linux, Solaris and Windows Vista as a method for combating exploits. Such an implementation requires the processor to permit memory segments to be designated as non-executable (newer X86 processors and processors belonging to the SPARC family support this feature). A non-executable stack is constructed by setting the no-execute (NX) bit for the appropriate memory segments. When a memory segment has the NX bit set, the processor refuses to execute data as instructions in the memory segment. The result is a Harvard-like memory organization where a section of memory is designated as non-executable data space. While the address spaces are not entirely separate, the characteristics of the two segments in memory loosely equate to the program and data address spaces encountered in the Harvard architecture.

Attackers have developed methods for defeating non-executable memory segments by utilizing available functionality. Specifically, instead of pushing custom instructions into the machine, execution is directed to pre-existing functionality. The strategy of using existing code to circumvent no-execute protection is leveraged in a `return-to-libc` attack [12]. This type of attack manipulates the stack so that it is intact and functional when the return pointer redirects execution to a function within the program. Exploit writers have used the `return-to-libc` attack to cause programs to invoke remote shells, down-

load toolkits and execute commands on a host system even when the system has non-executable memory enabled. The `return-to-libc` attack is analogous to identifying functionality within the data space of a Harvard class machine and causing the program to return to the desired functionality with the stack in a valid state. An example is redirecting execution to an update routine used to change data in embedded system memory. The routine is already present in the machine; however, clever manipulation of the stack enables the attacker to upload a new ROM image of his/her choosing.

7. Defense Techniques

Several techniques have been devised to guard against buffer overflows, most of them involve strict bounds checking or surrounding buffers with verifiable values. The values placed on the stack are called "canaries" (from the old practice of using birds to detect deadly gas in coal mines). To detect an overflow, a variable with a known value is placed before and after the variables in a stack frame. If an overflow occurs, the variable placed after the buffer is overwritten and its value is, therefore, altered. Before the executing function exits, it checks the value of this canary variable. The program is terminated if the value has changed, thwarting an attempt at forcing a redirected function return.

Several canary implementations have been devised. Systems used in von Neumann architectures, such as the randomized canaries employed by StackGuard [3] and ProPolice [13], protect stack frames and mitigate attacks that redirect execution. StackGuard modifies the GNU C compiler so that an integer value is inserted just before the return pointer in each stack frame and another copy of the integer is placed before the local variables. Comparing the two values before function return helps determine if an overflow has occurred, at which point the program is terminated. ProPolice, used in BSD and the GNU C version 4.1 compiler, incorporates a traditional random canary system and reorganizes the placement of variables within frames. By reorganizing variables and placing buffers at the beginning of frames, the compiler makes it more difficult to overwrite return pointers without destroying critical data.

Canary systems are not limited to protecting entire frames and detecting overflows. The Keil C compiler prevents overflows from occurring by performing a bounds check on every write to a buffer; this involves an integrity check of the null byte located at the end of the buffer. To accomplish this, the compiler translates array assignments into complete function calls that perform the bounds check and determine if the write is allowed. This method works well for null terminated strings. However, if a program allows nulls to be written to a buffer, an attacker can simply write a null byte to the location of the check value. The check routine would interpret the null value as an intact canary and permit the write. While this method is effective for simple character array operations and null terminated strings, it is not appropriate for all systems. The Keil C canary method also dramatically increases the overhead of write operations. A normal translation of an array write produces five to seven machine instructions. Since the Keil C compiler translates the array operation

into a complete function call, it produces ten to fifteen instructions, including several (slow) memory operations.

Mitigating overflows using canaries comes with a significant performance penalty. The security of a randomized canary is based on the statistical improbability of correctly guessing its value; this requires the invocation of a random number generation routine, which takes time. Canary values also consume stack space and require extra instructions to be executed before and after function calls. Other canary systems that protect individual buffers instead of entire frames involve considerable overhead. Small systems that rely on buffered input/output can be dramatically slowed by the decreased speed of operations on buffers. Interestingly, the Keil C compiler is targeted specifically at small embedded systems, which are usually input/output driven and rely on fast asynchronous interaction with external devices.

8. Conclusions

Stack-based buffer overflow vulnerabilities existing in the Harvard architecture can be exploited in much the same manner as in von Neumann systems. However, the process of exploiting a vulnerability is tricky, and delivering or identifying an exploit payload can be relatively complicated.

The memory management model used in the Harvard architecture changes the direction of writes in relation to stack growth; this simplifies the task of obtaining control of the instruction path. Memory space limitations may require compiler writers to be frugal with stack allocation, often leading to the clustering of key variables. In the case of the Keil C compiler, the non-standard placement of global variables permits direct access to the complete list of return pointers, which actually simplifies the exploit. The separation of data and program space in the Harvard architecture limits the ability of an attacker to inject operational code, but it does not limit the ability to redirect execution. Once an attacker has control of execution, he/she can find code to execute as in the case of a `return-to-libc` attack. Embedded systems commonly have the ability to dynamically update their code. The dynamic update systems are used for remote patching, feature updates and general fixes. Such a system can be exploited, for example, by using a buffer overflow to enter an update routine and then introduce unauthorized code into the embedded device.

Exploit writers have traditionally focused on powerful desktop and server systems, but attackers are increasingly targeting smaller devices, including embedded systems based on the Harvard architecture. Generalized protection mechanisms such as canaries are effective, but come with significant overhead that can impact the real-time performance of embedded systems, especially those used to control critical infrastructure assets.

Acknowledgements

This research was partially supported by NSF Scholarship for Service (SFS) Grant DUE 0621348.

References

[1] Aleph One, Smashing the stack for fun and profit, *Phrack*, vol. 49(14), 1996.

[2] J. Anderson, Computer Security Technology Planning Study, ESD-TR-73-51, Vol. 1, Deputy for Command and Management Systems, HQ Electronic Systems Division, United States Air Force, Hanscom Field, Bedford, Massachusetts, 1972.

[3] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Burke, S. Beattie, A. Grier, P. Wagle and Q. Zhang, StackGuard: Automatic adaptive detection and prevention of buffer overflow attacks, *Proceedings of the Seventh USENIX Security Symposium*, pp. 63–78, 1998.

[4] S. Eisenbach, *Functional Programming: Languages, Tools and Architectures*, Ellis Horwood, Chichester, United Kingdom, 1987.

[5] A. Francillon and C. Castelluccia, Code injection attacks on Harvard-architecture devices, *Proceedings of the Fifteenth ACM Conference on Computer and Communications Security*, pp. 15–26, 2008.

[6] Q. Gu and R. Noorani, Towards self-propagate mal-packets in sensor networks, *Proceedings of the First ACM Conference on Wireless Network Security*, pp. 172–182, 2008.

[7] N. Hanebutte and P. Oman, An evaluation of static source code analyzers for automated vulnerability detection, *Proceedings of the Ninth IASTED International Conference on Software Engineering and Applications*, pp. 112–117, 2005.

[8] N. Hanebutte and P. Oman, Software vulnerability mitigation as a proper subset of software maintenance, *Journal of Software Maintenance and Evolution*, vol. 17(6), pp. 379–400, 2006.

[9] Hitex, C51 Primer: An Introduction to the Use of the Keil C51 Compiler on the 8051 Family, Coventry, United Kingdom (www.hitex.com/fileadmin/img/download/c51_primer_290404.pdf), 2004.

[10] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta and R. Hassell, *The Shellcoder's Handbook*, Wiley, Indianapolis, Indiana, 2004.

[11] D. Seeley, A tour of the worm, *Proceedings of the Winter USENIX Conference*, pp. 287–304, 1989.

[12] Solar Designer, `return to-libc attack`, *Bugtraq*, 1997.

[13] R. Stallman, Using the GNU Compiler Collection, GNU Press, Boston, Massachusetts (gcc.gnu.org/onlinedocs/gcc-4.3.3/gcc.pdf), 2003.

[14] J. Waclawek, The unofficial history of 8051 (www.efton.sk/t0t1/history 8051.pdf), 1996.

[15] W. Wolf, *Computers as Components: Principles of Embedded Computing System Design*, Morgan Kaufmann, San Diego, California, 2001.